Programming Techniques: Project

An Implementation of Hunt the Wumpus

Andrew Noon

Introduction

This report details an implementation of the classic computer game Hunt the Wumpus, written in C, with the addition of watchable computer-controlled wumpus-hunter. After a brief overview of the history behind the game, the report outlines the design and implementation of the game from original specification to code. Then, a guide explaining how to play the game from a user's point of view is presented, followed by details of testing and debugging procedures that were carried out to ensure that the implementation was both functional and robust.

Background

Hunt the Wumpus, in its original incarnation, was an archetypal dungeon exploration game. Written first in BASIC by Gregory Yob, it came to the fore most notably in 1975 when its code and an accompanying description were published in Creative Computing magazine. Hunt the Wumpus differed from the mainly grid-based adventure games of the day by having the eponymous creature reside in a dodecahedral cave of twenty rooms, and featured Yob's characteristic sense of humour throughout the narration.

Following its publication Hunt the Wumpus grew quickly in popularity, being rewritten in almost every programming language available, and produced for many of the popular platforms of the past and present. Such was its popularity during the computer game boom of the 1970s that the monstrous yet mysterious wumpus has become iconic in computer science, appearing, as a knowing nod and tribute, in multiple games developed since, such as NetHack.

Specification

The implementation of Hunt the Wumpus described here adheres to Gregory Yob's classic BASIC program; whilst the structure of the code itself has not been translated directly into C, the functionality of the game has been kept as faithful as possible. Where changed it has been noted in this report, and the reasons for doing so will be explained.

Hunt the Wumpus falls into the genre of adventure game. An adventure game features the player traversing an environment, usually in the presence of numerous obstacles and/or opponents, exploring, gaining knowledge, and ultimately seeking their way to the goal state. The environment in Hunt the Wumpus is the labyrinthine cave of twenty rooms, arranged as the vertices of a dodecahedron. Each room has three tunnels leading to other rooms; these tunnels are two-way and can be utilised in either direction.

The wumpus is an unidentified monster which resides in the twenty-room cave; the slaying of it results in successful completion of the game. The wumpus is a decidedly slumberous creature, spending most of its time asleep in one of twenty rooms that constitute its lair. Should the player enter the room in which the wumpus is sleeping, it will wake and either eat the player or escape to another room. A player so eaten is deemed to have lost the game. If the wumpus should decide to make good its escape, the hunt continues with another turn for the player.

Along with the mobile wumpus, the cave also features two other types of static hazard. Bottomless pits occur in two of the twenty rooms; upon entering a room containing a bottomless pit, the player will fall to their death, ending the game. In another two rooms are superbats, which will transport an unsuspecting player to another room in the cave at random. This may well have the undesired side-effect of dropping a player into a pit, or onto the sleeping wumpus. Unlike the hunter, the wumpus moves around its cave independently of

these hazards. His sucker feet enable to him traverse the bottomless pits safely and he is far too large for the superbats to transport.

The player interacts with the game as though it were he or she exploring the cave; there is no avatar. In the case of the computer-controlled agent, narration is given of a nameless wumpus-hunter. For brevity, this report will refer to the cave explorer as the hunter, regardless of whether it is under player or computer control. The hunter carries with him (or her) a bow capable of firing magical arrows with which to slay the wumpus. Magical arrows are capable of navigating a predetermined path through up to five rooms before running out of energy and falling to the floor. Should the arrow's path take it through a room in which the wumpus is asleep, it will kill the wumpus and win the game for the hunter. However, the arrow can also kill an unwary hunter who directs it to his own room, thus resulting in a loss. The number of arrows is limited (traditionally five) and should the hunter expend his last arrow without successfully hitting the wumpus, the game is also lost. As with entering its room, the unsuccessful firing of an arrow also wakes the wumpus; upon waking, it will either go back to sleep or decide to move rooms. Clearly this can have dire consequences for any wumpus-hunters in adjoining rooms.

The game consists of the hunter taking turns to either move through the cave one tunnel at a time, or shoot an arrow. Nothing else occurs to the cave or its denizens without prior action by the hunter. The pits and superbats remain in the same rooms throughout the course of a game (the bats presumably returning to their roost after depositing their cargo). At the start of each turn the hunter is given clues as to what may be found in adjoining rooms in the form of short warnings. The wumpus is an odorous creature and can therefore be smelled one room away in all directions. Superbats possess such large wings that they can be heard from the next room, and bottomless pits cause a flow of air that can be felt from a similar distance.

The game, as defined above, should be functionally identical for both human and computer hunters. Upon loading the game, the user should be able to specify whether to control the hunter or watch the computer do the same. It is specified explicitly that the computer controlled agent should play the game as if it were a human player. This is to say that it should have no access to information other than that which is presented on-screen. It cannot cheat and refer to the internal cave map that the game produces to keep track of game objects.

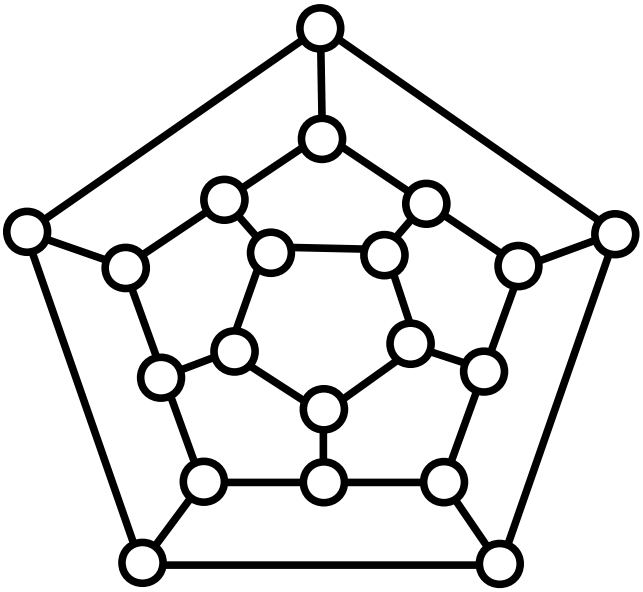
Implementation

The game is implemented in a single C file. Despite being approximately 400 lines of code in length, the decision was made to keep the program in a single file for ease of implementation. There is no necessity to split functions into a separate library and import them as it is not expected that these functions will see use outside of this particular program. Build time throughout the project was consistently low (under half a second) so continually revising one single file presented no issues. In larger projects (e.g. more than 1000 lines of code) it would be advantageous to split code that is expected to remain unchanged into separate files so that only that code which is actively being worked upon needs to be recompiled.

Wherever possible, judicious use of functions has been employed to reduce code redundancy. This is especially notable where both human and computer controlled hunters employ very similar routines to navigate their way through the game. For the same reason, the decision-making process of the computer-controlled hunter is isolated within separate functions. This will be covered in more detail below when discussing specific functions.

The cave is represented by a two-dimensional array with dimensions 20 by 3, representing the interconnectivity of tunnels for the default cave parameters. No information about the contents of each room is held within the array. An alternative implementation would be to define a room structure and construct a one-dimensional array of these rooms; however given the limited amount of information held about each room this level of complexity was deemed unnecessary. All of the relevant game objects (player, wumpus, hazards) can be tracked equally well by their own independent location variables. One issue that needed

careful attention throughout implementation was array numbering beginning at zero. Rooms 1-20 as they are identified to the user, were therefore numbered 0-19 as far as the cave array was concerned.



A two-dimensional representation of the dodecahedral cave.

Due to the dodecahedral nature of the cave, its layout is hard-coded into the program; the numbers of interconnected rooms remain constant from game to game to aid the player's attempts to mentally map the cave. With future expansion in mind, the dimensions of the cave are defined as named constants at the start of the file and then referred to by `ROOMS` and `EXITS` for the remainder of the code. In this way new cave layouts can be easily introduced providing a fixed network of tunnels is defined; all changes to the code in doing this are conveniently confined to the first handful of declarations. The number of arrows the hunter starts with and the maximum number of rooms through which they can be fired are defined similarly so these can be adjusted without changes to the main body of code. Win and lose conditions are defined as 2 and 1 respectively to make for more readable code; these are covered in more detail below.

The `main()` function of the C file manages the main menu choices of the program. These are whether the player would like to control the hunter themselves or watch the computer play, and also whether to start a new game or exit having finished a previous game. Note that it is assumed having started the program that the player will want to play at least one game; the option to exit formally is only presented after this first game is completed. Progress of the game itself depends wholly on two functions called from within `main()`: `play_game()` and `watch_game()`. These functions progress the game in similar ways, differing mainly in terms of input and output. Both make use of a while loop that checks `result` to see if the game has either been won or lost. If no result has yet been achieved the function checks if the hunter has arrows remaining (an instant lose condition if not true) and then calls `check_wumpus()` and `check_hazard()` which are outlined below. If either function returns a 1, it indicates that a game-ending condition has been encountered and the rest of the turn should be skipped. To ease the user in watching a computer-controlled hunter, the game is paused after each turn (or after each room of an arrow's path when the computer decides to shoot). This is achieved by prompting the user to press any key, and using the `getch()` function to catch this; the character entered is not assigned to any variable. A third function could easily be implemented to enable a player and computer-controlled hunter to coexist within the cave, combining the functionality of both `play_game()` and `watch_game()`.

Player choice within `main()` is handled by the compiler extension `getch()`. This takes the first character entered without the need to press return thus creating a more responsive user interface. This function is used throughout the remainder of the code where a single character is expected. Input of integers that may be double digit (e.g. room numbers) is handled by `scanf()` imported from the C standard library. With all

input throughout the program, both the prompt and input catching functions are contained within infinite loops that are only broken upon entrance of a valid response.

One considerable code-saving feature is the use of pointers to individual pieces of narrative text; these are declared at the start of the file with meaningful names e.g. `txt_hitwump`. Two different flavours of string are then defined for each item of narrative text; one for human players, and one for the computer-controlled wumpus-hunter. The functions `set_playertext()` and `set_cputext()` are called to reassign the pointers based on whether the player or the computer is controlling the hunter. All other text is deliberately worded to remain non-committal about the identity of the protagonist i.e. “There is a slight draft” as opposed to “You feel a slight draft”. It would be equally valid to strip out all such hard-coded text and have this declared at the head of the file, but for simplicity only those where ambiguity was unattainable were so defined in this project. One advantage to having all in-game text defined independently of the code would be that this text could be imported from an external file. These files could be interchangeable and one could switch languages, settings or themes with the underlying functionality of the game remaining the same e.g. nuclear bunker for cave, terminator for wumpus, homing missiles for magical arrows.

The function `warn()` is used during games by both human and computer hunters. Firstly it checks whether the wumpus is in a room adjoining the hunter and prints the relevant narration if this is the case. If the wumpus is present the variable `smell_wumpus` is set to 1 to indicate that a computer-controlled hunter is aware of the wumpus' scent. This is used later in the agent's decision-making process outlined below.

In a straightforward expansion to the classic specification of Hunt the Wumpus presented above, additional warnings indicate if an unlucky hunter finds himself in a room surrounded by both bats or both pits. This is achieved by stepping through the array of exits for the hunter's room, and incrementing local variables `num_bats` and `num_pits` in the presence of either hazard. As explained above, the narration is kept deliberately ambiguous so the function can be reused for both human and computer hunters.

Player moves are carried out by the function `player_move()`, which is distinct from `cpu_move()` outlined below. This is the first example of function separation between player and computer hunters, and was implemented in this fashion as much of the `player_move()` function consists of output and input validation. Despite being twenty lines in length all that is in fact required for the function to take effect on the game world is one of three integers indicating a valid move. The location of the hunter is then set to this position (after correcting for array notation beginning at zero).

The presence of a wumpus in the room is checked at the start of each turn by the function `check_wumpus()`. If the hunter's location matches that of the wumpus then the wumpus will wake and either eat the hunter or flee to another room, each of these occurring with a 50% probability. If the wumpus should decide to defend itself and eats the hunter, then global variable `result` is set to LOSE. This has the effect of breaking the while loop in `play_game()` or `watch_game()` and causing the program to return to `main()`. If the wumpus decides to escape, it can move to each of the adjoining rooms with equal probability.

As with `check_wumpus()`, `check_hazards()` also sets `result` to LOSE if a pit is encountered. If a superbat is encountered then the hunter is transported to one of the twenty rooms with equal probability. Note that this includes the possibility that the hunter is transported back to the room in which he was picked up, or to the other superbat's room. There is therefore a small but definite possibility that the hunter could be endlessly transported around the cave from bat to bat. This could be eliminated with the addition of a simple while loop, but seems in keeping with the spirit of the game and was retained in the program to emphasise the randomness of encountering such a superbat.

The firing of magical arrows is covered in two separate functions `player_shoot()` and `cpu_shoot()`. As with `player_move()` and `cpu_move()`, two separate functions are necessary to eliminate much of the input validation required for human players. Both versions of the shoot function call two subfunctions `fire_arrow()` and `arrow_wumpus()` which deal with arrow collision detection and wumpus waking

respectively. The first of these functions is called for each room that the arrow passes through to see if it hits either the wumpus or the hunter. If it hits the former then `result` is set to WIN and a 1 is returned in order to break the for loop that steps though the arrow's path. Similarly, if the arrow should enter the same room as the hunter, striking him, then `result` is set to LOSE and again a 1 is returned to break out of the the shoot routine.

Functions `cpu_move()` and `cpu_shoot()` are both called by `cpu_decide()`. Much as its name would suggest, this function decides what the computer-controlled hunter will do as it makes its way around the cave. The agent is implemented fairly simply as moving around the cave at random until it smells a wumpus (as indicated by the variable s`mell_wumpus`). The variable `last_room` is tracked indicating the room that the agent vacated most recently. This ensures that the agent does not retrace its steps whilst roaming the cave trying to locate a scent. When it does locate the wumpus' smell, it will fire an arrow through one of its current room's tunnels, excluding the one which it has just entered via (again tracked with `last_room`). The arrow then travels on for a random number of rooms ensuring it avoids both the hunter and the last room exited (which is known to be wumpus-free).

Implementation of the computer-controlled agent in this way enables much of the code to be reused efficiently. It also increases the modularity of the program, meaning more or less advanced agent routines can be introduced easily without impacting the rest of the program. Three main functions dictate the play of the computer-controlled hunter and these can be replaced independently or together to achieve different behaviours e.g. a hunter that infers the positions of game objects and maps the cave meticulously, but still fires at random. Barring an unlucky initial cave layout, any sufficiently advanced agent would be expected to be successful almost 100% of the time, so in the interests of compelling gameplay a more naïve agent is perhaps desirable.

Initialisation of the cave is carried out by the `random_cave()` function. This is distinct from the function `test_cave()` discussed below under testing. The hunter and wumpus are placed in different rooms, and the hazards are placed ensuring they do not clash with the hunter or one another. The random placement of each game object is carried out within its own while loop; whilst this makes for slightly more verbose code, computationally it means that multiple comparisons and calls to `rand()` are kept to minimum.

User Manual

The program can be run from the command line or as a Windows executable. Upon entering the game the user is presented with the title screen and an option to either play a game themselves, or watch the computer do the same. The user makes this decision by pressing either 'p' to play, or 'w' to watch; note there is no necessity to press return. The program will indicate if an invalid selection has been made.

Upon starting a human-controlled game, the user is presented with information detailing their location, remaining arrows, and any clues to their surroundings. They can then opt to move or shoot by pressing the relevant key, and again the program will correct them if the wrong key is pressed. If they choose to move they are given a choice of tunnels which they can take; these are entered by typing the number of the room and pressing return. Again, the program ensures that the user enters a valid room before proceeding. If the user opts to shoot, they are given the option of how many rooms to fire the arrow through. By default this is up to five but can be tweaked to any amount so desired by the programmer, and therefore input requires typing the relevant integer and pressing return. Having selected the number of rooms to which the arrow will fly, the user is then presented with a choice of tunnels the arrow can take given its present position. After each selection, the outcome of the arrow's flight is calculated and the results are displayed on-screen.

Should the player be successful in slaying the wumpus, or the player should fall foul to the wumpus, a pit, or arrow, the game is over and after a short explanatory message the player is asked if they wish to play another game of either type, or exit the program.

Watching a computer-controlled hunter differs in that the only input required by the user is to step through the hunter's turns until the game has been resolved. This is achieved by pressing any key; there is no need to enter any specific commands.

Testing/Debugging

Testing was aided by the addition of functions `output_cave()` and `test_cave()`. The first of these outputs the interconnectivity of the cave and the locations of all inhabitants. This can be used mainly to ensure that `random_cave()` is setting up the initial game state according to specification, but also to speed up later test runs where the effects of particular hazards are being sought. The second function is used in a similar fashion to locate all hazards in fixed locations to speed up repeated test encounters with them.

Care was taken throughout the programming process to maintain the distinction between the zero-based array notation in the code, and the one-based numbering of the cave rooms, whilst narrating the hunter's progress. This was deemed early on as an easy mistake to make, but one that could have disastrous consequences for any prospective wumpus-hunters acting on the displayed information.

The main bulk of testing and bug-finding was carried out in the spirit of the game, by allowing various users to play as it neared completion. Whilst debugging code is paramount, the testing process must also cover the intention for the game to be played by a variety of users with a range of technical proficiencies, so this method of testing was deemed more suitable than a similar programmer-based or automated method.

One error that remains in the program, and that could be fixed in future implementations, is the entry of non-integers to calls of `scanf()`. The use of `getch()` was preferred where possible to avoid this, but in cases where a double-digit integer may be required this was impossible.

Bibliography

Yob, G. (1976) 'Hunt the Wumpus', The Best of Creative Computing, Volume 1

Appendix – wumpus.c

```
/*
 * wumpus.c
 *
 *  Last edit:        12-Jan-2008
 *  Author:    Andrew Noon
 *
 *  An implementation of the Hunt the Wumpus game. *
 *
 *  TODO: variable numbers of bats and pits
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define WIN 2
#define LOSE 1
/* define cave system parameters */
#define ROOMS 20
#define EXITS 3
#define MAX_ARROWS 5 /* arrows that player starts with */
#define MAX_RANGE 5 /* maximum range of magical arrow */

/* construct layout of dodecahedral cave system */
static int cave[ROOMS][EXITS] = {
               {1,4,7},{0,2,9},{1,3,11},{2,4,13},{0,3,5},{4,6,14},{5,7,16},{0,6,8},
               {7,9,17},{1,8,10},{9,11,18},{2,10,12},{11,13,19},{3,12,14},{5,13,15},
               {14,16,19},{6,15,17},{8,16,18},{10,17,19},{12,15,18}
};
/* variables tracking positions of various game items */
int loc_hunter, loc_wump, loc_bat1, loc_bat2, loc_pit1, loc_pit2;
int arrows; /* number of arrows remaining */
int smell_wump = 0; /* can cpu hunter smell wumpus? */
int last_room; /* room that cpu hunter just exited */
int result; /* result of individual game */
int exit_game = 0; /* set to 1 to terminate program */
int i, j; /* loop indexes */

char *txt_eat;
char *txt_pit;
char *txt_loc;
char *txt_wumpwarn;
char *txt_grab;
char *txt_hitwump;
char *txt_hitself;

char txt_eat_p[] = "The wumpus wakes and eats you!\n";
char txt_eat_c[] = "The wumpus wakes and eats the wumpus-hunter!\n";
char txt_pit_p[] = "You have fallen in a pit!\n";
char txt_pit_c[] = "The wumpus-hunter has fallen in a pit!\n";
char txt_loc_p[] = "\nYou are in room %d\n";
char txt_loc_c[] = "\nThe wumpus-hunter is in room %d\n";
char txt_wumpwarn_p[] = "You sense the fetid stench of a wumpus\n";
char txt_wumpwarn_c[] = "The wumpus-hunter smells a wumpus!\n";
char txt_grab_p[] = "A superbat drags you to a different room!\n";
char txt_grab_c[] = "A superbat drags the wumpus-hunter to another room!\n";
char txt_hitwump_p[] = "\nYour arrow hits the wumpus, slaying it!\n";
char txt_hitwump_c[] = "\nThe hunter's arrow hits the wumpus, slaying it!\n";
char txt_hitself_p[] = "\nYou've been struck by your own arrow!\n";
char txt_hitself_c[] = "\nThe hunter has been struck by his own arrow!\n";

int getch(void); /* from standard library */

void set_playertext(void) {
       txt_eat = txt_eat_p;
       txt_pit = txt_pit_p;
       txt_loc = txt_loc_p;
       txt_wumpwarn = txt_wumpwarn_p;
       txt_grab = txt_grab_p;
       txt_hitwump = txt_hitwump_p;
       txt_hitself = txt_hitself_p;
}

void set_cputext(void) {
       txt_eat = txt_eat_c;
       txt_pit = txt_pit_c;
       txt_loc = txt_loc_c;
```

```
        txt_wumpwarn = txt_wumpwarn_c;
        txt_grab = txt_grab_c;
        txt_hitwump = txt_hitwump_c;
        txt_hitself = txt_hitself_c;
}

/* warns player of relevant hazards in adjoining rooms */
void warn(void) {
        /* check adjoining rooms for wumpus */
        for (i=0; i<EXITS; i++) {
                if (cave[loc_hunter][i] == loc_wump) {
                        printf(txt_wumpwarn);
                        smell_wump = 1;
                        break;
                } else smell_wump = 0;
        }
        /* count hazards in adjoining rooms */
        int num_bats = 0;
        int num_pits = 0;
        for (i=0; i<EXITS; i++) {
                 if (cave[loc_hunter][i] == loc_bat1) num_bats++;
                 if (cave[loc_hunter][i] == loc_bat2) num_bats++;
                 if (cave[loc_hunter][i] == loc_pit1) num_pits++;
                 if (cave[loc_hunter][i] == loc_pit2) num_pits++;
        }
        /* output text according to surround bats */
        if (num_bats == 2) printf("There are bats all around\n");
        else if (num_bats == 1) printf("It would seem bats are nearby\n");
        else printf("There is no sign of bats\n");
        /* output text according to surrounding pits */
        if (num_pits == 2) printf("It is very drafty\n");
        else if (num_pits == 1) printf("There is a slight draft\n");
        else printf("The air is still\n");
}

/* moves the player from room via relevant tunnels */
void player_move(void) {
        int move_room;
        int valid_move = 0;
        printf("\nThere are tunnels leading to rooms:");
        for (i=0; i<EXITS; i++) {
                printf(" %d", cave[loc_hunter][i]+1);
        }
        do {
                printf("\nTo which room would you like to move? ");
                scanf("%d", &move_room);
                for (i=0; i<EXITS; i++) {
                        if (move_room == cave[loc_hunter][i]+1) {
                                valid_move = 1; /* is a valid move */
                        }
                }
                if (valid_move == 0) {
                        printf("\nThere is no tunnel leading to that room\n");
                }
        } while (valid_move == 0);
        printf("\nYou trudge down the tunnel to room %d\n", move_room);
        loc_hunter = move_room-1;
}

/* checks for the presence of a wumpus and wakes it if necessary */
int check_wumpus(void) {
        if (loc_hunter == loc_wump) {
                /* decide if wumpus moves or defends itself
                 * TODO: tweakable probability for wumpus' reaction */
                if (rand() % 2 == 1) {
                        printf(txt_eat);
                        result = LOSE;
                        return 1;
                } else {
                        printf("The wumpus wakes and runs to the next room\n");
                        /* decides which room the wumpus moves to */
                        loc_wump = cave[loc_wump][rand() % EXITS];
                        return 0;
                }
        } else return 0;
}
```

```
/* checks for hazards and carries out any necessary actions upon the player */
int check_hazards(void) {
	if (loc_hunter == loc_bat1 || loc_hunter == loc_bat2) {
		printf(txt_grab);
		loc_hunter = rand() % ROOMS;
		return 1;
	} else if (loc_hunter == loc_pit1 || loc_hunter == loc_pit2) {
		printf(txt_pit);
		result = LOSE;
		return 1;
	} else
		return 0;
}

/* checks if the arrow has hit anything */
int fire_arrow(int target) {
	if (target == loc_wump) {
		printf(txt_hitwump);
		result = WIN; return 1;
	} else if (target == loc_hunter) {
		printf(txt_hitself);
		result = LOSE; return 1;
	} else return 0;
}

/* checks wumpus' action once arrow has been fired */
void arrow_wumpus(void) {
	/* decide if wumpus moves upon waking
	 * TODO: tweakable probability for wumpus' reaction */
	if (rand() % 2 == 1) {
		printf("\nSomething large moves in the distance\n");
		/* decides which room the wumpus moves to */
		loc_wump = cave[loc_wump][rand() % EXITS];
	}
}

/* performs an arrow shot */
int player_shoot(void) {
	arrows--;
	int loc_arrow, arrow_range, target, valid_shot;
	while (1) {
		printf("\nHow many rooms do you wish the magical arrow "
				"to travel through? (1-%d)\n", MAX_RANGE);
		scanf("%d", &arrow_range);
		if (0 < arrow_range && arrow_range <= MAX_RANGE) {
			break;
		} else printf("\nInvalid number of rooms. Please choose again\n");
	}
	loc_arrow = loc_hunter; /* arrow begins in room with player */
	for (i=0; i<arrow_range; i++) {
		valid_shot = 0; /* reset valid shot */
		printf("\nThere are tunnels leading to rooms:");
		for (j=0; j<EXITS; j++) {
			printf(" %d", cave[loc_arrow][j]+1);
		}
		do {
			printf("\nTo which room should your arrow travel? ");
			scanf("%d", &target);
			for (j=0; j<EXITS; j++) {
				if (target-1 == cave[loc_arrow][j]) {
					valid_shot = 1; /* is a valid move */
					break;
				}
			}
			if (valid_shot == 0) {
				printf("\nThere is no tunnel leading to that room\n");
			}
		} while (valid_shot == 0);
		loc_arrow = target-1;
		if (fire_arrow(target-1)) return 1;
		printf("\nYour arrow continues on its trajectory\n");
	}
	printf("\nYour arrow runs out of magical energy and falls to the floor\n");
	arrow_wumpus();
	return 0; /* arrow hits nothing */
}
```

```
void cpu_move(void) {
	int destination;
	do { /* ensure cpu hunter doesn't backtrack */
		destination = cave[loc_hunter][rand() % EXITS];
	} while (destination == last_room);
	last_room = loc_hunter;
	loc_hunter = destination;
	printf("\nThe hunter trudges down the tunnel to room %d\n", loc_hunter+1);
}

int cpu_shoot(void) {
	printf("\nThe hunter fires a magical arrow from his bow\n");
	arrows--;
	int loc_arrow, arrow_range, target;
	arrow_range = (rand() % MAX_RANGE) + 1;
	loc_arrow = loc_hunter; /* arrow begins in room with hunter */
	for (i = 0; i < arrow_range; i++) {
		/* hunter does not fire into room just exited or own room */
		do {
			target = cave[loc_arrow][rand() % EXITS];
		} while (target == last_room || target == loc_hunter);
		loc_arrow = target;
		printf("\nThe arrow flies into room %d\n", target+1);
		if (fire_arrow(target))
			return 1;
		printf("...but hits nothing\n");
		printf("\nPress any key to continue\n");
		getch();
	}
	printf("\nThe arrow runs out of magical energy and falls to the floor\n");
	arrow_wumpus();
	return 0; /* arrow hits nothing */
}

void cpu_decide(void) {
	if (smell_wump) {
		cpu_shoot();
	} else cpu_move();
}

/* randomise initial cave setup */
void random_cave(void) {
	loc_hunter = rand() % ROOMS; /* randomise starting position of player*/
	/* randomly place wumpus in separate room from player */
	do {
		loc_wump = rand() % ROOMS;
	} while (loc_wump == loc_hunter);
	/* randomly place hazards in separate rooms from player and each other */
	do {
		loc_bat1 = rand() % ROOMS;
	} while (loc_bat1 == loc_hunter);
	do {
		loc_bat2 = rand() % ROOMS;
	} while (loc_bat2 == loc_hunter || loc_bat2 == loc_bat1);
	do {
		loc_pit1 = rand() % ROOMS;
	} while (loc_pit1 == loc_hunter || loc_pit1 == loc_bat1
			|| loc_pit1 == loc_bat2);
	do {
		loc_pit2 = rand() % ROOMS;
	} while (loc_pit2 == loc_hunter || loc_pit2 == loc_bat1
			|| loc_pit2 == loc_bat2 || loc_pit2 == loc_pit1);
}

/* test cave for debug purposes */
void test_cave(void) {
	/* fixes player, wumpus and hazards one move away for testing */
	loc_hunter = 0; loc_wump = 1;
	loc_bat1 = 4; loc_bat2 = 3;
	loc_pit1 = 7; loc_pit2 = 6;
}
```

```
void output_cave(void) {
	for (i=0; i<ROOMS; i++) {
				printf("Room %d has exits: ", i+1);
				for (j=0; j<EXITS; j++) { printf("%d, ", cave[i][j]+1); }
				printf("\n");
		}
	printf("P: %d W: %d ", loc_hunter+1, loc_wump+1);
	printf("B1: %d B2: %d ", loc_bat1+1, loc_bat2+1);
	printf("P1: %d P2: %d\n", loc_pit1+1, loc_pit2+1);
}

void play_game(void) {
	set_playertext();
	system("cls");
	result = 0;
	arrows = MAX_ARROWS;
	while (!result) {
		if (!arrows) {
			printf("\nYou have run out of magical arrows!\n");
			break;
		}
		if (check_wumpus()) continue; /* check for wumpus */
		if (check_hazards()) continue; /* check for hazards */
		printf("\nYou are in room %d\n", loc_hunter+1);
		printf("Magical arrows remaining: %d\n\n", arrows);
		warn(); /* print any relevant warnings */
		while(1) {
			printf("\nWould you like to (m)ove or (s)hoot an arrow?\n");
			char turn = getch();
			if (turn == 'm') {
				player_move();
			} else if (turn == 's') {
				player_shoot();
			} else {
				printf("Invalid choice!\n");
				continue;
			}
			break;
		}
	}
	if (result == WIN) {
		printf("\nYou have won! Congratulations!\n");
	} else printf("\nGame over!\n");
}

void watch_game(void) {
	set_cputext();
	system("cls");
	result = 0;
	arrows = MAX_ARROWS;
	while (!result) {
		if (!arrows) {
			printf("\nThe wumpus-hunter has run out of magical arrows!\n");
			break;
		}
		if (check_wumpus()) continue; /* check for wumpus */
		if (check_hazards()) continue; /* check for hazards */
		printf("\nThe wumpus-hunter is in room %d\n", loc_hunter+1);
		printf("Magical arrows remaining: %d\n\n", arrows);
		warn(); /* print any relevant warnings */
		while(1) {
			printf("\nPress any key to continue\n");
			getch();
			cpu_decide();
			break;
		}
	}
	if (result == WIN) {
		printf("\nThe wumpus-hunter has won!\n");
	} else printf("\nGame over!\n");
}
```

```
int main() {
	srand(time(NULL)); /* seed RNG */
	while(!exit_game) {
		system("cls");
		printf("---------- HUNT THE WUMPUS ----------\n");
		while (1) {
			printf("\nDo you want to (p)lay, or (w)atch the computer play?\n");
			char choice = getch();
			if (choice == 'p') {
				random_cave();
				/*test_cave();
				output_cave();*/
				play_game();
				break;
			} else if (choice == 'w') {
				random_cave();
				/*test_cave();
				output_cave();*/
				watch_game();
				break;
			} else printf("Invalid choice!\n");
		}
		while(1) {
			printf("\nWould you like to start a (n)ew game or (e)xit?\n");
			char reset = getch();
			if (reset == 'n') {
				break;
			} else if (reset == 'e') {
				exit_game = 1;
				break;
			} else printf("Invalid choice!\n");
		}
	}
	return 0; /* program terminated successfully */
}
```